Memory & data
- Integers & floats
- Machine code & C

x86 assembly

Procedures & stacks

Arrays & structs

Memory & caches

Processes

Virtual memory

Memory allocation

Java vs. C

---

### Roadmap

**C:**
```c
#include "car.h"

int main()
{
    Car* car = new Car();
    car->miles = 100;
    car->gals = 17;
    float mpg = get_mpg(car);
    delete car;
    return 0;
}
```

**Java:**
```java
public class Car {
    int miles;
    int gals;

    public void setMiles(int miles) {
        this.miles = miles;
    }

    public void setGals(int gals) {
        this.gals = gals;
    }

    public float getMPG() {
        return miles / gals;
    }
}
```

**Assembly language:**
```
get_mpg:
    pushq   %rbp
    movq    %rsp, %rbp
    ...
    popq    %rbp
    ret
```

**Machine code:**
```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

**Computer system:**

Windows 8

Mac

- Processor
- Memory
- Hard drive
Procedures and Call Stacks

- How do I pass arguments to a procedure?
- How do I get a return value from a procedure?
- Where do I put local variables?
- When a function returns, how does it know where to return?

- To answer these questions, we need a call stack ...
Memory Layout

- **Stack**: local variables; procedure context
- **Dynamic Data (Heap)**: variables allocated with `new` or `malloc`
- **Static Data**: `static` variables (including global variables (C))
- **Literals**: literals (e.g., “example”)
- **Instructions**

Memory Addresses

- **High Addresses**: $2^{N-1}$
- **Low Addresses**: 0
Memory Layout

- Stack: writable; not executable
  - Managed “automatically” (by compiler)

- Dynamic Data (Heap): writable; not executable
  - Managed by programmer

- Static Data: writable; not executable
  - Initialized when process starts

- Literals: read-only; not executable
  - Initialized when process starts

- Instructions: read-only; executable
  - Initialized when process starts

segmentation faults?
IA32 Call Stack

- Region of memory managed with a stack “discipline”
  - Grows toward lower addresses
  - Customarily shown “upside-down”

- Register $\%esp$ contains lowest stack address
  = address of “top” element

Stack Pointer: $\%esp$
IA32 Call Stack: Push

- `pushl Src`

Stack Pointer: `%esp`

Stack "Bottom"

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses

0x00...00
IA32 Call Stack: Push

- **pushl** *Src*
  - Fetch value from *Src*
  - *Decrement* $\%\text{esp}$ by 4 (*why 4?*)
  - Store value at address given by $\%\text{esp}$

- **Example:**
  - **pushl** $\%\text{ecx}$
  - Adjust $\%\text{esp}$ and store contents of $\%\text{ecx}$ on the stack
IA32 Call Stack: Pop

- popl Dest

Stack Pointer: \%esp

Stack “Bottom”

High Addresses

Increasing Addresses

Stack Grows Down

Low Addresses 0x00...00

Stack “Top”
IA32 Call Stack: Pop

- `popl Dest`
  - Load value from address `%esp`
  - Write value to `Dest`
  - **Increment** `%esp` by 4

Example:

- `popl %ecx`
  - Stores contents of top of stack into `%ecx` and adjust `%esp`

**Stack Pointer:** `%esp`
IA32 Call Stack: Pop

- popl Dest
  - Load value from address %esp
  - Write value to Dest
  - Increment %esp by 4

Stack Pointer: %esp

Those bits are still there; we’re just not using them.
### Procedure Call Overview

- **Callee** must know where to find `args`
- **Callee** must know where to find “return address”
- **Caller** must know where to find return val
- **Caller** and **Callee** run on same CPU → use the same registers
  - So how do we deal with register reuse?

---

**Diagram:**

- **Caller**
  - `...`
  - `<set up args>
  - call`
  - `<clean up args>
  - `<find return val>
  - `...`

- **Callee**
  - `<create local vars>
  - `...`
  - `<set up return val>
  - `<destroy local vars>
  - `return`
The **convention** of where to leave/find things is called the calling convention (or procedure call linkage).

- Details vary between systems
- We will see the convention for IA32/Linux in detail
- What could happen if our program didn’t follow these conventions?
Procedure Control Flow

- Use stack to support procedure call and return
- **Procedure call**: `call label`
  1. Push return address on stack (*why?*, and *which exact address?*)
  2. Jump to `label`
Procedure Control Flow

- Use stack to support procedure call and return

**Procedure call:** `call label`

1. Push return address on stack
2. Jump to `label`

**Return address:**
- Address of instruction after `call`
- Example from disassembly:

```
804854e:   e8 3d 06 00 00       call  8048b90 <main>
8048553:   03 45 08           addl  0x08(%ebp),%eax
```

Return address = `0x8048553`

**Procedure return:** `ret`

1. Pop return address from stack
2. Jump to address

Next instruction just happens to be an add, but could be anything
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>804854e:</td>
<td>e8 3d 06 00 00</td>
<td>call 8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>8048553:</td>
<td>03 45 08</td>
<td>addl 0x08(%ebp),%eax</td>
</tr>
</tbody>
</table>

%eip: program counter
%

%esp: 0x108
%

%eip: 0x804854e
Procedure Call Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Instruction</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x804854e:</td>
<td>e8 3d 06 00 00</td>
<td>call</td>
<td>8048b90 &lt;main&gt;</td>
</tr>
<tr>
<td>0x8048553:</td>
<td>03 45 08</td>
<td>addl</td>
<td>0x08(%ebp),%eax</td>
</tr>
</tbody>
</table>

**Diagram:**
- `call 8048b90`
- `%esp`: 0x108
- `%eip`: 0x804854e
- `%esp`: 0x104
- `%eip`: 0x8048553
- `0x108`: 123
- `0x10c`: (Blank)
- `0x110`: (Blank)
- `0x104`: 0x8048553

**Notes:**
- `%eip`: Program counter

*Autumn 2014*
Procedure Call Example

```plaintext
804854e:  e8 3d 06 00 00  call  8048b90  <main>
8048553:  03 45 08  addl  0x08(%ebp),%eax
```

- **%esp**: Program counter
- **%eip**: Relative address

Relative address just like jumps...
(chosen by compiler; there's also an absolute call)
Procedure Return Example

\begin{center}
\begin{tabular}{|c|c|}
\hline
8048591: & c3 \ \ \ \ \ \ \ \ \ \ \ \ \ \ ret \\
\hline
\end{tabular}
\end{center}

\textbf{\textcolor{red}{%eip}: program counter}

\textbf{\textcolor{green}{%esp}: stack pointer}

\textbf{\textcolor{blue}{ret}}
Procedure Return Example

8048591: c3 ret

%esp 0x104
0x110
0x10c
0x108 123
0x104 0x8048553

%eip 0x8048591

%eip: program counter

ret

%esp 0x104
0x110
0x10c
0x108 123
0x104 0x8048553

%eip 0x8048591
Procedure Return Example

8048591: c3
ret

ret
0x110
0x10c
0x108
0x104
0x110
0x10c
0x108
0x104
%esp 0x104
%esp 0x104
%eip 0x8048591
%eip 0x8048591
%eip 0x8048591
%eip 0x8048591

%eip: program counter

Autumn 2014
Procedures and Stacks
Procedure Return Example

8048591:  c3  ret

%esp  0x104
%eip  0x8048591

%esp  0x104  0x8048553
%esp  0x108  123
%esp  0x10c
%esp  0x110
%esp  0x110
%eip  0x8048591
%eip  0x8048553

ret

%eip: program counter
IA32 Return Values

- By convention, values returned by procedures are placed in the %eax register
  - Choice of %eax is arbitrary, could have easily been a different register
- **Caller** must make sure to save the contents of %eax before calling a **callee** that returns a value
  - Part of register-saving convention
- **Callee** places return value into the %eax register
  - Any type that can fit in 4 bytes – integer, float, pointer, etc.
  - For return values greater than 4 bytes, best to return a pointer to them
- Upon return, **caller** finds the return value in the %eax register
Stack-Based Languages

- Languages that support recursion
  - e.g., C, Java, most modern languages
  - Code must be re-entrant
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

- Stack discipline
  - State for a given procedure needed for a limited time
    - Starting from when it is called to when it returns
  - Callee always returns before caller does

- Stack allocated in frames
  - State for a single procedure instantiation
Call Chain Example

```
who(...) {
    .
    amI();
    .
}
```

```
who(...) {
    .
    .
}
```

```
yoo(...) {
    .
    who();
    .
    .
}
```

```
amI(...) {
    .
    .
    if () {
        amI()
    }
    .
}
```

Procedure amI is recursive (calls itself)
Stack Frames

Contents
- Function arguments
- Return address
- Local variables
- Temporary space

Management
- Space allocated when procedure is entered
  - “Set-up” code
- Space deallocated upon return
  - “Finish” code
Example:

call to `yoo`

```c
yoo (...) {
  // ...
  who();
  // ...
}
```

Stack:

- `%ebp`
- `%esp`

`yoo`
Example:

**call to who**

```c
who(...)  
{  
  •  
    ami();  
  •  
    ami();  
  •  
}  
```

Stack:

- yoo
- who

%ebp
%esp
Example:

**call to `amI`**

```
amI(...) {
  •
  if() {
    amI()
  }
  •
}
```

Stack:

- `%ebp`
- `%esp`
- `yoo`
- `who`
- `amI`
Example:

recursive call to `amI`

```
amI(...) {
  if () {
    amI()
  }
}
```
Example:

(another) recursive call to `amI`

```
amI(...) {
    .
    if() {
        amI()
    }
    .
}
```

Stack:

```
%ebp
%esp
```

- `yoo`
- `who`
- `amI`
- `amI`
- `amI`
Return from:
(another) recursive call to `amI`

```
amI(...) {
  if() {
    amI()
  }
}
```

Stack

```
%ebp
%esp
amI
amI
amI
amI
```

`yoo`
Return from:

recursive call to amI

amI(...) {
    •
    if() {
        amI()
    }
    •
}

Stack

yoo

who

%ebp

%esp

amI

amI

amI

amI

amI
Return from:
call to amI

who(...)  
{        
  •      
  amI(); 
  •      
  amI(); 
}        

yoo   
|      
|      
who   
|      
|      
amI  
|      
amI  
amI  
amI

Stack

yoo
%ebp
%esp
amI
amI
amI
amI
Example:

(Second) call to amI

```cpp
amI (...) {
  .
  if() {
    amI()
  }
  .
}
```

Stack

```
Stack
  yoo
  who
  amI
  %ebp
  %esp
amI
```
Example from:

(second) call to `amI`

```
who(...) {
  .
  amI();
  .
  amI();
}
```

Stack:

```
%ebp
%esp
```

```
yoo
who
amI
amI
amI
amI
```
Return from:
call to who

\[
\text{yoo(...)}
\{
\quad \cdot
\quad \cdot
\quad \text{who();}
\quad \cdot
\quad \cdot
\}
\]

How did we remember where to point %ebp when returning?
IA32/Linux Stack Frame

- **Caller’s Stack Frame**
  - Arguments for this call
  - Return address
    - Pushed by `call` instruction

- **Current /Callee Stack Frame**
  - Old frame pointer (for caller)
  - Saved register context (when reusing registers)
  - Local variables (if can’t be kept in registers)
  - “Argument build” area (if callee needs to call another function - parameters for function about to be called)
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```c
void call_swap()
{
    pushl $zip2    # Global Var
    pushl $zip1    # Global Var
    call swap
    ...
}
```

we know the address

```c
call_swap:
    ...
```
Revisiting swap

```c
int zip1 = 15213;
int zip2 = 98195;

void call_swap()
{
    swap(&zip1, &zip2);
}

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
```

Calling swap from call_swap

```c
call_swap:
    ... ...
    pushl $zip2   # Global Var
    pushl $zip1   # Global Var
    call swap
    ... ...
```

Resulting Stack

- Rtn adr
- &zip1
- &zip2
- %esp
Revisiting swap

void swap(int *xp, int *yp)
{
    int t0 = *xp;
    int t1 = *yp;
    *xp = t1;
    *yp = t0;
}
swap Setup, step #1

Entering Stack

Resulting Stack?

\[
\begin{align*}
\text{swap:} & \quad \text{pushl} \ %\text{ebp} \\
& \quad \text{movl} \ %\text{esp},%\text{ebp} \\
& \quad \text{pushl} \ %\text{ebx} \\
& \quad \text{Set Up}
\end{align*}
\]
swap Setup, step #1

Entering Stack

Resulting Stack

swap:

\[
\begin{align*}
\text{pushl } \%ebp \\
\text{movl } \%esp, \%ebp \\
\text{pushl } \%ebx \\
\end{align*}
\]

Set Up

%ebp

%esp

0x100

0x060

&zip2

&zip1

Rtn adr

0x100

0x060

yp

xp

Rtn adr

Old %ebp

%ebp

%esp
swap Setup, step #2

Entering Stack

0x100

•

•

•

&zip2

&zip1

Rtn adr

%ebp

%esp

0x060

Resulting Stack

0x100

•

•

•

yp

xp

Rtn adr

Old %ebp

%ebp

%esp

0x060

swap:

pushl %ebp

movl %esp,%ebp

pushl %ebx

Set Up

%ebp

%esp
swap Setup, step #3

swapping:

pushl %ebp
movl %esp,%ebp
pushl %ebx

Set Up

Entering Stack

0x100

%ebp

•

•

•

&zip2

&zip1

Rtn adr

0x060

%esp

Resulting Stack

0x100

•

•

•

yp

xp

Rtn adr

Old %ebp

Old %ebx

0x060

%ebp

%esp

%esp

%ebp
swap Body

**Entering Stack**

<table>
<thead>
<tr>
<th>$0x100$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$%ebp$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;zip2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;zip1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rtn adr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$0x060$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$%esp$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Resulting Stack**

<table>
<thead>
<tr>
<th>$0x100$</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>yp</td>
<td>xp</td>
<td>Rtn adr</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>8</th>
<th>Old $%ebp$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Old $%ebx$</td>
</tr>
</tbody>
</table>

---

```
movl 12(%ebp), %ecx  # get yp
movl 8(%ebp), %edx   # get xp
```

Body
swap Finish, step #1

Finishing Stack

Resulting Stack?

```assembly
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```

Finish
swap Finish, step #1

Observation: The Callee saved and restored register `%ebx`

Finishing Stack

Resulting Stack

```
movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret
```
swap Finish, step #2

Finishing Stack

Resulting Stack

\[
\begin{align*}
\text{movl} & \ -4(\%ebp),\%ebx \\
\text{movl} & \ \%ebp,\%esp \\
\text{popl} & \ \%ebp \\
\text{ret} & \\
\end{align*}
\]

\text{Finish}
swap Finish, step #3

Finishing Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Finish
swap Finish, step #4

Finishing Stack

Resulting Stack

movl -4(%ebp),%ebx
movl %ebp,%esp
popl %ebp
ret

Finish
Disassembled swap

080483a4 <swap>:

```
80483a4:   55          push   %ebp
80483a5:   89 e5       mov    %esp,%ebp
80483a7:   53          push   %ebx
80483a8:   8b 55 08    mov    0x8(%ebp),%edx
80483ab:   8b 4d 0c    mov    0xc(%ebp),%ecx
80483ae:   8b 1a       mov    (%edx),%ebx
80483b0:   8b 01       mov    (%ecx),%eax
80483b2:   89 02       mov    %eax,(%edx)
80483b4:   89 19       mov    %ebx,(%ecx)
80483b6:   5b          pop    %ebx
80483b7:   c9          leave
80483b8:   c3          ret
```

Calling Code

```
8048409:   e8 96 ff ff ff   call 80483a4 <swap>
804840e:   8b 45 f8       mov    0xfffffffff8(%ebp),%eax
```

relative address (little endian)
swap Finish #4

Observation
- Saved & restored register %ebx
- but not %eax, %ecx, or %edx

Finishing Stack
- %ebp
- %esp
- Old %ebp
- Old %ebx
- Rtn adr
- xp
- yp

Resulting Stack
- %ebp
- %esp
- yp
- xp

movl \(-4(\%ebp)\),%ebx
movl %ebp,%esp
popl %ebp
ret
Register Saving Conventions

- **When procedure** yoo **calls** who:
  - yoo is the **caller**
  - who is the **callee**

- **Can a register be used for temporary storage?**

  ```asm
  yoo:
  ...
  movl $12345, %edx
  call who
  addl %edx, %eax
  ...
  ret
  ```

  ```asm
  who:
  ...
  movl 8(%ebp), %edx
  addl $98195, %edx
  ...
  ret
  ```

  - Contents of register %edx overwritten by who
Register Saving Conventions

- When procedure \texttt{yoo} calls \texttt{who}:
  - \texttt{yoo} is the \textit{caller}
  - \texttt{who} is the \textit{callee}

- Can a register be used for temporary storage?

- Conventions
  - \textit{“Caller Save”}
    - Caller saves temporary values in its frame before calling
  - \textit{“Callee Save”}
    - Callee saves temporary values in its frame before using
IA32/Linux Register Usage

- **%eax, %edx, %ecx**
  - **Caller** saves prior to call if values are to be used after returning from the call
  - %eax also used to return integer value

- **%ebx, %esi, %edi**
  - **Callee** saves if wants to use them

- **%esp, %ebp**
  - special form of **callee** save – restored to original values upon exit from procedure
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x) {
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

Recursive Procedure

```c
void s_helper(int x, int *accum) {
    if (x <= 1) {
        return;
    } else {
        int z = *accum * x;
        *accum = z;
        s_helper(x-1, accum);
    }
}
```

sfact(3)

Pass pointer to update location
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```c
sfact(3)  val = 1
s_helper(3, &val)
```

Recursive Procedure

```c
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper(x-1,accum);
    }
}
```

Pass pointer to update location
Example: Pointers to Local Variables

Top-Level Call

```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```
sfact(3)       val = 1
s_helper(3, &val)  val = 3
s_helper(2, &val)  
```

Recursive Procedure

```
void s_helper
    (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

Pass pointer to update location
Example: Pointers to Local Variables

Top-Level Call

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

Recursive Procedure

```c
void s_helper
  (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

```c
void s_helper
  (int x, int *accum)
{
    if (x <= 1)
        return;
    else {
        int z = *accum * x;
        *accum = z;
        s_helper (x-1,accum);
    }
}
```

sfact(3) val = 1
s_helper(3, &val) val = 3
s_helper(2, &val) val = 6
s_helper(1, &val)

Pass pointer to update location
Creating & Initializing Local Var (1)

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it

Initial part of `sfact`

```
_sfact:
    pushl %ebp          # Save %ebp
    movl %esp,%ebp     # Set %ebp
    subl $16,%esp      # %esp=%esp-16bytes
    movl 8(%ebp),%edx  # %edx = x
    movl $1,-4(%ebp)   # val = 1
```

<table>
<thead>
<tr>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td><code>%esp</code></td>
</tr>
<tr>
<td><code>%edx</code></td>
</tr>
<tr>
<td><code>x</code></td>
</tr>
</tbody>
</table>

Note:
- Variable `val` must be stored on stack because it needs to be created with a pointer to it.
Creating & Initializing Local Var (2)

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it

### Initial part of `sfact`

```
_sfact:
    pushl %ebp # Save %ebp
    movl %esp,%ebp # Set %ebp
    subl $16,%esp # %esp=%esp-16bytes
    movl 8(%ebp),%edx # %edx = x
    movl $1,-4(%ebp) # val = 1
```

```
\begin{array}{|c|}
\hline
8 & x \\
4 & Rtn adr \\
0 & Old %ebp \\
-4 & \%ebp \\
-8 & Temp. Space \\
-12 & %esp \\
-16 & \\
\hline
\end{array}
```
Creating & Initializing Local Var (3)

```
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it

Initial part of `sfact`

```
_sfact:
    pushl %ebp      # Save %ebp
    movl %esp,%ebp  # Set %ebp
    subl $16,%esp   # %esp=%esp-16bytes
    movl 8(%ebp),%edx # %edx = x
    movl $1,-4(%ebp) # val = 1
```

```
<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>Rtn adr</th>
<th>Old %ebp</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0</td>
<td>%ebp</td>
</tr>
<tr>
<td>-4</td>
<td></td>
<td>-4</td>
<td>val = 1</td>
</tr>
<tr>
<td>-8</td>
<td></td>
<td>-8</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td></td>
<td>-12</td>
<td>Unused</td>
</tr>
<tr>
<td>-16</td>
<td></td>
<td>-16</td>
<td></td>
</tr>
</tbody>
</table>
```

- %ebp
- %esp
Passing Pointer to Local Val (1)

int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}

- Variable val must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as $-4\ (\%ebp)$
- Push on stack as second argument

Calling s_helper from sfact

```
leal -4(%ebp),%eax # Compute &val
pushl %eax # Push on stack
pushl %edx # Push x
call s_helper # call
movl -4(%ebp),%eax # Return val
    ... # Finish
```
Passing Pointer to Local Val (2)

```c
int sfact(int x)
{
    int val = 1;
    s_helper(x, &val);
    return val;
}
```

- Variable `val` must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as `-4 (%ebp)`
- Push on stack as second argument

Stack at time of call:

```
<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>x</td>
</tr>
<tr>
<td>4</td>
<td>Rtn adr</td>
</tr>
<tr>
<td>0</td>
<td>Old %ebp</td>
</tr>
<tr>
<td>-4</td>
<td>val = 1</td>
</tr>
<tr>
<td>-8</td>
<td>Unused</td>
</tr>
<tr>
<td>-12</td>
<td>&amp;val</td>
</tr>
<tr>
<td>-16</td>
<td>x</td>
</tr>
</tbody>
</table>
```

Calling `s_helper` from `sfact`

```
leal -4(%ebp),%eax  # Compute &val
pushl %eax          # Push on stack
pushl %edx          # Push x
call s_helper       # call
movl -4(%ebp),%eax  # Return val
      ...         # Finish
```
IA 32 Procedure Summary

Important points:

- IA32 procedures are a combination of instructions and conventions
  - Conventions prevent functions from disrupting each other
- Stack is the right data structure for procedure call / return
  - If P calls Q, then Q returns before P

Recursion handled by normal calling conventions

- Can safely store values in local stack frame and in callee-saved registers
- Put function arguments at top of stack
- Result returned in %eax
x86-64 Procedure Calling Convention

- Doubling of registers makes us less dependent on stack
  - Store argument in registers
  - Store temporary variables in registers

- What do we do if we have too many arguments or too many temporary variables?
## x86-64 64-bit Registers: Usage Conventions

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
<tr>
<td>%rbx</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%rcx</td>
<td>Argument #4</td>
</tr>
<tr>
<td>%rdx</td>
<td>Argument #3</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument #2</td>
</tr>
<tr>
<td>%rdi</td>
<td>Argument #1</td>
</tr>
<tr>
<td>%rsp</td>
<td>Stack pointer</td>
</tr>
<tr>
<td>%rbp</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>%r8</td>
<td>Argument #5</td>
</tr>
<tr>
<td>%r9</td>
<td>Argument #6</td>
</tr>
<tr>
<td>%r10</td>
<td>Caller saved</td>
</tr>
<tr>
<td>%r11</td>
<td>Caller Saved</td>
</tr>
<tr>
<td>%r12</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r13</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r14</td>
<td>Callee saved</td>
</tr>
<tr>
<td>%r15</td>
<td>Callee saved</td>
</tr>
</tbody>
</table>
Revisiting swap, IA32 vs. x86-64 versions

swap:
- pushl %ebp
- movl %esp,%ebp
- pushl %ebx
- movl 12(%ebp),%ecx
- movl 8(%ebp),%edx
- movl (%ecx),%eax
- movl (%edx),%ebx
- movl %eax,(%edx)
- movl %ebx,(%ecx)
- movl -4(%ebp),%ebx
- movl %ebp,%esp
- popl %ebp
- ret

swap (64-bit long ints):
- movq (%rdi), %rdx
- movq (%rsi), %rax
- movq %rax, (%rdi)
- movq %rdx, (%rsi)
- retq

- Arguments passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required (except ret)
- Avoiding stack
  - Can hold all local information in registers
X86-64 procedure call highlights

- Arguments (up to first 6) in registers
  - Faster to get these values from registers than from stack in memory
- Local variables also in registers (if there is room)
- Registers still designated “caller-saved” or “callee-saved”
- `callq` instruction stores 64-bit return address on stack
  - Address pushed onto stack, decrementing `%rsp` by 8
- No frame pointer
  - All references to stack frame made relative to `%rsp`; eliminates need to update `%ebp/%rbp`, which is now available for general-purpose use
- Functions can access memory up to 128 bytes beyond `%rsp`: the “red zone”
  - Can store some temps on stack without altering `%rsp`
x86-64 Memory Layout

- **Static Data**
- **Dynamic Data (Heap)**
- **Literals**
- **Instructions**
- **Stack**

The **128-byte red zone** space lower than the stack pointer that procedures can use for data not needed across procedure calls. This optimization avoids extra `%rsp` updates.
x86-64 Stack Frames

- Often (ideally), x86-64 functions need no stack frame at all
  - Just a return address is pushed onto the stack when a function call is made

- A function *does* need a stack frame when it:
  - Has too many local variables to hold in registers
  - Has local variables that are arrays or structs
  - Uses the address-of operator (&) to compute the address of a local variable
  - Calls another function that takes more than six arguments
  - Needs to save the state of caller-save registers before calling a procedure
  - Needs to save the state of callee-save registers before modifying them
x86-64 Example (1)

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call Proc:

```assembly
subq $32,%rsp
movq $1,16(%rsp)  # x1
movl $2,24(%rsp)  # x2
movw $3,28(%rsp)  # x3
movb $4,31(%rsp)  # x4
• • •
```

Return address to caller of call_proc

NB: Details may vary depending on compiler.
x86-64 Example (2) – Allocate local vars

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
```assembly
    subq  $32, %rsp
    movq  $1, 16(%rsp)  # x1
    movl  $2, 24(%rsp)  # x2
    movw  $3, 28(%rsp)  # x3
    movb  $4, 31(%rsp)  # x4
    ... ...
```

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
<th>x1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>%rsp</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
long int call_proc()
{
    long  x1 = 1;
    int   x2 = 2;
    short x3 = 3;
    char  x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

Return address to caller of call_proc

<table>
<thead>
<tr>
<th>x4</th>
<th>x3</th>
<th>x2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

x1
<table>
<thead>
<tr>
<th>x1</th>
</tr>
</thead>
</table>

16
<table>
<thead>
<tr>
<th>Arg 8</th>
</tr>
</thead>
</table>

8
<table>
<thead>
<tr>
<th>Arg 7</th>
</tr>
</thead>
</table>

Arguments passed in (in order):
rdi, rsi, rdx, rcx, r8, r9

call proc

call_proc:
    • • •
    leaq  24(%rsp),%rcx  # %rcx=&x2
    leaq  16(%rsp),%rsi  # %rsi=&x1
    leaq  31(%rsp),%rax  # %rax=&x4
    movq  %rax,8(%rsp)   # arg8=&4
    movl  $4,(%rsp)      # arg7=4
    leaq  28(%rsp),%r9   # %r9=&x3
    movl  $3,%r8d        # %r8 = 3
    movl  $2,%edx        # %rdx = 2
    movq  $1,%rdi        # %rdi = 1
    call proc
    • • •
### x86-64 Example (4) – setup params to `proc`

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

#### return address to caller of `call_proc`

<table>
<thead>
<tr>
<th>Return address to caller of call_proc</th>
</tr>
</thead>
<tbody>
<tr>
<td>24</td>
</tr>
<tr>
<td>x4</td>
</tr>
<tr>
<td>x1</td>
</tr>
</tbody>
</table>

#### `call_proc`:

```assembly
• • •
leaq 24(%rsp),%rcx
leaq 16(%rsp),%rsi
leaq 31(%rsp),%rax
movq %rax,8(%rsp)
movl $4,(%rsp)
leaq 28(%rsp),%r9
movl $3,%r8d
movl $2,%edx
movq $1,%rdi
```

Arguments passed in (in order): `rdi, rsi, rdx, rcx, r8, r9`

Note sizes
long int call_proc() {
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2,
         x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}

call_proc:
    • • •
    leaq 24(%rsp),%rcx
    leaq 16(%rsp),%rsi
    leaq 31(%rsp),%rax
    movq %rax,8(%rsp)
    movl $4,(%rsp)
    leaq 28(%rsp),%r9
    movl $3,%r8d
    movl $2,%edx
    movq $1,%rdi
    call proc
    • • •
x86-64 Example (6) – after call to `proc`

```c
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:

```
    ; ; ;
    movswl 28(%rsp),%eax  # %eax=x3
    movsbl 31(%rsp),%edx  # %edx=x4
    subl   %edx,%eax  # %eax=x3-x4
    cltq
    movslq 24(%rsp),%rdx  # %rdx=x2
    addq   16(%rsp),%rdx  # %rdx=x1+x2
    imulq  %rdx,%rax  # %rax=rax*rdx
    addq   $32,%rsp
    ret
```

Return address to caller of call_proc

| 24 | 16 | 8 | 8%
|----|----|---|---|
| x4 | x3 | x2 | %rsp
| 24 | 16 | 8 |

**movs** – move and sign extend

**cltq** - sign extend %eax into %rax
x86-64 Example (7)

```
long int call_proc()
{
    long x1 = 1;
    int x2 = 2;
    short x3 = 3;
    char x4 = 4;
    proc(x1, &x1, x2, &x2, x3, &x3, x4, &x4);
    return (x1+x2)*(x3-x4);
}
```

call_proc:
    movswl 28(%rsp),%eax
    movsbl 31(%rsp),%edx
    subl %edx,%eax
    cltq
    movslq 24(%rsp),%rdx
    addq 16(%rsp),%rdx
    imulq %rdx,%rax
    addq $32,%rsp
    ret

Return address to caller of call_proc
x86-64 Procedure Summary

- Heavy use of registers (faster than using stack in memory)
  - Parameter passing
  - More temporaries since more registers

- Minimal use of stack
  - Sometimes none
  - When needed, allocate/deallocate entire frame at once
  - No more frame pointer: address relative to stack pointer

- More room for compiler optimizations
  - Prefer to store data in registers rather than memory
  - Minimize modifications to stack pointer